

Typed Answer Set Programming Lambda Calculus Theories and Correctness of Inverse Lambda Algorithms with respect to them

Chitta Baral, Juraj Dzifcak, Marcos A. Gonzalez and Aaron Gottesman

*School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ*

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Our broader goal is to automatically translate English sentences into formulas in appropriate knowledge representation languages as a step towards understanding and thus answering questions with respect to English text. Our focus in this paper is on the language of Answer Set Programming (ASP). Our approach to translate sentences to ASP rules is inspired by Montague’s use of lambda calculus formulas as meaning of words and phrases. With ASP as the target language the meaning of words and phrases are ASP-lambda formulas. In an earlier work we illustrated our approach by manually developing a dictionary of words and their ASP-lambda formulas. However such an approach is not scalable. In this paper our focus is on two algorithms that allow one to construct ASP-lambda formulas in an inverse manner. In particular the two algorithms take as input two lambda-calculus expressions G and H and compute a lambda-calculus expression F such that F with input as G , denoted by $F@G$, is equal to H ; and similarly $G@F = H$. We present correctness and complexity results about these algorithms. To do that we develop the notion of typed ASP-lambda calculus theories and their orders and use it in developing the completeness results.

KEYWORDS: Natural Language Understanding, Answer Set Programming, Lambda Calculus, Inverse Lambda Algorithms

1 Introduction

The broader goal of our proposed research is to translate English sentences to appropriate knowledge representation (KR) and reasoning languages. This will help in understanding text and answering questions with respect to it. Such an ability is important in developing various systems that need “understanding of natural language”. This includes systems that can acquire knowledge from text, systems that can interact in English with robots and other systems, intelligent training and tutoring systems, and systems that can process existing scientific literature in particular domains and formulate hypothesis.

Our approach is inspired by Montague’s work (Montague 1974) where the meaning of words and phrases are expressed as λ -calculus expressions and the meaning of a sentence is built from the meaning of its words by making appropriate appli-

cations of the corresponding λ -calculus expressions. This approach has also been used elsewhere, such as in (Blackburn and Bos 2005; Zettlemoyer and Collins 2005; Baral et al. 2008; Dzifcak et al. 2009; Costantini and Paolucci 2010; Baral et al. 2012); but the question that we address here is *how do we obtain the λ -calculus like meaning of words? They get complex quickly and hand crafting them is not scalable.*

In this paper we use ASP as our target KR language and address the issue of automatically obtaining meaning of words as ASP- λ -calculus formulas instead of the underlying logic of first order logic in traditional use of λ -calculus. Thus, the meanings of words are expressed as formulas of ASP- λ -calculus and using them sentences are translated to ASP rules. In (Baral et al. 2008) the ASP- λ -calculus formulas that represent words were handcrafted and it was remarked that the human engineering needed to generate the ASP- λ -calculus expressions need to be substituted by an automatic process.

Our main idea in automating this process is through a *a notion of inverse application of ASP- λ -calculus formulas* and use them in constructing the ASP- λ -calculus expressions of words. We discuss two algorithms from (Gonzalez 2010) that compute a ASP- λ -calculus expression¹ F given ASP- λ -calculus expressions G and H . In the first algorithm, which we call the $Inverse_L$ algorithm, the F is such that by applying G as an input to it one obtains H ; this is written as $F@G = H$. In the second algorithm, which we call the $Inverse_R$ algorithm, F is such that $G@F = H$. We refer to these algorithms as the Inverse λ -Algorithms. In this paper we define ASP- λ -calculus formulas and formalize how this approach can be used to translate words into these formulas, yielding a method to automatically translate sentences into ASP rules.

We illustrate the basic idea of inverse application of ASP- λ -calculus formulas and how they can be used in constructing the ASP- λ -calculus expressions of words through the following example.

	Most	birds	fly
	$(S/(S \backslash NP))/NP$	NP	$S \backslash NP$
	$S/(S \backslash NP)$		$S \backslash NP$
			S
Most	birds	fly	
???	$\lambda x.bird(x)$	$\lambda x.fly(x)$	
???		$\lambda x.fly(x)$	
	$fly(X) \leftarrow bird(X), not \neg fly(X)$		

Table 1. CCG and λ -calculus derivation for “Most birds fly”.

It is assumed in Table 1 that the meaning of “Most birds fly” and the ASP- λ -calculus formulas for “fly” and “birds” is known. We would like to determine the appropriate semantic representation for the word “most”. To do so we must first compute the semantic representation of “Most birds”. This can be done using the meaning of the sentence “Most birds fly” and the word “fly”. However, we first

¹ This algorithm also works for typed first-order logic lambda calculus. We show that in (Baral et al. 2012). But its applicability to ASP is not discussed there as that requires additional machinery, which we present in this paper.

must know whether the meaning of “Most birds” is to be used as input to the meaning of “fly” or vice versa. To obtain this directionality information we make use of combinatory categorial grammars (CCG) (Steedman 2000).

A CCG parse of a sentence assigns categories to the words of the sentence. There are several basic categories, with S representing a sentence and NP representing a noun phrase. More complex categories are formed from these basic categories by using “\” and “/” which specify directionality. For example, a non-transitive verb, like “fly” above, would have category $S \backslash NP$ meaning that if a noun phrase, NP , precedes the verb then a sentence S is formed. Similarly, a category for a simple adjective, would be NP / NP , meaning that if a noun phrase, NP , comes after the adjective then a NP would result.

Note that the category of “most” given here is not that of a simple adjective, NP / NP . If “most” had this category then the result of applying “birds” to “most” would result in category NP which would then be applied to the right of “fly”. However, it is not possible to form the meaning of the sentence by substituting into the given meaning of “fly”. Therefore, an alternative CCG parse of the sentence must be used that swaps the application of “fly” to be on the right side. This is done by raising the category of “Most birds” to $S / (S \backslash NP)$, which in turns means that the category of “most” must be $(S / (S \backslash NP)) \backslash NP$.

The top part of Table 1 gives a CCG parse of the sentence “Most birds fly”. The meaning of the phrase “Most birds”, which has a category $S / (S \backslash NP)$, must have the meaning of “fly” applied from the right since it has the category $S \backslash NP$. Therefore, to get the meaning of the sentence, H , we let G be the meaning of “fly”. Then we have to find an F such that $F @ G = H$. From inspection $F = \lambda x. (x @ X \leftarrow \text{bird}(X), \text{not } \neg x @ X)$ will satisfy this equation.

Now, having the expressions for “Most birds” and “birds”, we can calculate the meaning of the desired word “most”. Since “most” has category $(S / (S \backslash NP)) \backslash NP$, we have to apply the meaning of “birds” to the right of it to obtain the meaning of “Most birds”. From inspection taking the meaning of “most” to be $\lambda v. \lambda x. (x @ X \leftarrow v @ X, \text{not } \neg x @ X)$ produces the desired result.

As this example demonstrates, given the meaning of most words in a sentence and a CCG parse for the sentence, we can find a new semantic representation for words and phrases whose meanings are unknown. The question is how exactly do we determine the new representation? In this paper we discuss the Inverse- λ Algorithms (Gonzalez 2010) to solve this task, which is known as the *Inverse- λ problem*.²

To help in showing the correctness and applicability of our $Inverse_L$ and $Inverse_R$ algorithms we *extrapolate the notion of typed λ first order theories to define the*

² The Inverse- λ problem can be shown to be a special cases of the “higher-order matching” (Dowek 1994) and “Interpolation problem” (Stirling 2009). Specifically, the $Inverse_L$ problem corresponds to an Interpolation problem and $Inverse_R$ problem corresponds to the Higher-order matching problem. The higher order matching problem is known to be undecidable in the general case (Loader 2003). Higher order matching can be further considered as a special case of higher order unification which has been explored in (Huet 1973; Huet 1975) and recently used in (Kwiatkowski et al. 2010). None of these works consider ASP.

notion of a typed ASP- λ theory. We then define the notion of orders of such theories. Using these notions we present the soundness, completeness and complexity results of the Inverse- λ Algorithms. For example, the completeness result is with respect to typed ASP- λ -calculus formulas up to the second order³. We then illustrate the use of the Inverse- λ Algorithms with respect to typed ASP- λ -calculus formulas.

As mentioned earlier, these algorithms are key to developing systems that can translate English sentences to KR languages. However, such systems need to address additional issues such as dealing with possible multiple meaning of words, and developing appropriate ontologies that maximize the accuracy of the translation. These aspects are separately discussed in (Dzifcak et al. 2009). A simpler version of the Inverse- λ Algorithms discussed in this paper is used in developing a system that learns to translate combinatorial puzzles to ASP rules and solve those puzzles (Baral and Dzifcak 2012)

In summary the main contributions of this paper are:

- We formulate the notion of typed ASP- λ -calculus theories and define the notion of orders of such theories.
- We illustrate the use of Inverse- λ Algorithms with respect to typed ASP- λ -calculus formulas.
- We present soundness, completeness and complexity results for these algorithms with respect to typed ASP- λ -calculus theories.

The rest of this paper is organized as follows. In the next section, we present some background material and pointers on typed lambda calculus and ASP. In Section 3 we introduce typed Answer Set Programming lambda calculus. In Section 4 we present the Inverse λ -Algorithms. We then illustrate our algorithms with respect to several examples and give a use of our algorithms in sections 5 and 6, respectively. In Section 7 we present the soundness, completeness and complexity results. Finally, we conclude and briefly mention the companion natural language semantics work that uses our algorithms.

2 Background

2.1 Typed Lambda Calculus

Since Montague’s groundbreaking work (Montague 1974), λ -calculus has been accepted and used as a tool by many in natural language semantics. Montague was the first to introduce the use of λ -calculus to represent the meaning of words and λ -application as a mechanism to construct the meaning of phrases and sentences. However, to properly understand the notion of “meaning” (or semantics), it is useful to consider models of λ -calculus expressions. When referring to a model, one is looking for a semantic tool that can give it two elements: the entities that are part

³ Blackburn and Bos say in page 101 of their book (Blackburn and Bos 2005): “Now, arguably natural language semantics never requires types much above order three or so—nonetheless the ability to take a logical perspective on higher-order types really is useful.” Note that their definition of order three corresponds to our definition of order 2.

of the domain, and for every element in the signature, the semantic value associated with it. By creating this model with the corresponding denotations for types, expressions of the system will have a defined type and a semantic value associated with it.

Both untyped and typed λ -calculus can be characterized using models, but typed λ -calculus has had the most impact on natural language semantics, which also became familiar to linguistics after the mentioned works by Montague. In this paper we will follow the *Simply Typed Lambda Calculus* of Church (Church 1940) to have ASP as the core logic. This is the most commonly used approach in linguistics where only one type constructor is used to build types, " \rightarrow ", and each term has a single type associated with it (Barendregt 1992).

Because of space constraints, we do not present the typed lambda calculus definitions that we use to define typed ASP- λ calculus. The books (Hindley 1986) and (Hindley 1997) are good reference points for typed lambda calculus.

2.2 Answer Set Programming

Answer Set Programming is the language of logic programming with answer set semantics (Gelfond and Lifschitz 1988). This language is one of the most suitable declarative languages (Baral 2003) for knowledge representation, reasoning and declarative problem solving; all important aspects that are needed to develop natural language understanding systems. It has a large body of support structure, including efficient implementations and theoretical building block studies. It allows the representation, in an intuitive way, of various kinds of knowledge that cannot be adequately expressed in first-order logic. These include, for instance, default statements (*most birds fly*) and normative statements (*normally birds fly*). We now present some basic definitions related to Answer Set Programming syntax and semantics (Baral 2003).

Definition 1 (ASP rule)

An *ASP rule* is of the form:

$$L_0 \text{ or } \dots \text{ or } L_k \leftarrow L_{k+1}, \dots, L_m, \text{ not } L_{m+1}, \dots \text{ not } L_n.$$

where L_i are literals and $k \geq 0$, $m \geq k$ and $n \geq m$.

The literals to the left of the " \leftarrow " belong to the *Head* of the rule, and the literals to the right of the " \leftarrow " belong to the *Body* of the rule. An *ASP program* is a set of *ASP rules*.

Definition 2 (Satisfiability)

An *ASP rule* of the form:

$$L_0 \text{ or } \dots \text{ or } L_k \leftarrow L_{k+1}, \dots, L_m, \text{ not } L_{m+1}, \dots \text{ not } L_n.$$

of an ASP program Π is said to be *satisfied* by a set of ground literals I of Π if:

- $\{L_{k+1}, \dots, L_m\} \subseteq I$ and $\{L_{m+1}, \dots, L_n\} \cap I = \emptyset$ implies that $\{L_0, \dots, L_k\} \cap I \neq \emptyset$.

Definition 3

An *Answer Set* of an ASP Program Π without the “not” operator, is a consistent set of ground literals S such that S satisfies all rules of Π and no subset of S satisfies all rules of Π .

Definition 4 (Answer Set)

A consistent set S of ground literals is an *Answer Set* of an ASP Program Π if S is an answer set of the reduct Π^S , where Π^S is obtained from Π by

- (i) Deleting all rules from Π that contain some *not* l in their body for some $l \in S$.
- (ii) Removing all occurrences of *not* l from the remaining rules.

3 Typed Answer Set Programming Lambda Calculus

We start by presenting the signature for the language Typed Answer Set Programming Lambda Calculus (Typed ASP Lambda Calculus). It consists of the following:

- the lambda operator, also called abstractor, λ ;
- the lambda application $@$;
- the parenthesis symbols $(,), [, \text{ and }]$;
- for every type a , an infinite set of variables $v_{n,a}$ for each natural number n ;
- for every type a , a (possibly empty) set of constants c_a of type a ;
- the connectives *or*, \leftarrow , \neg , *not*, “,” and “.”; and
- predicate and function symbols.

Variables and constants in the signature for Typed ASP Lambda Calculus will be referred to as λ -terms.

Next, we introduce the set of types that will be used with Typed ASP Lambda Calculus, in conjunction with the definition of the semantics of types assigned to the different expressions of the language. We will follow the principles presented in (Barbara H. Partee and Wall 1990), where D_a represents the set of possible objects (*denotations*) that describe the meanings of expressions of type a .

Definition 5 (Types)

The *set of types* Θ is defined recursively as follows:

1. e, a, l, g, d, h, t are types, called *base types*, and
2. if A and B are types, then $(A \rightarrow B)$ is a type.

Intuitively, e refers to terms, which is either a variable or a constant in ASP, or a function symbol with terms as input; a refers to atoms of ASP, which are predicate symbols with terms as input⁴; l refers to literals of ASP which are atoms or atoms preceded by the connective \neg ; g refers to *gen-literals* which are literals or literals preceded by the connective *not*; d refers to a conjunction of gen-literals, where the conjunction is denoted by “,”; h refers to a disjunction of literals, where the disjunction is denoted by “or”; and t refers to the boolean truth values. More formally,

⁴ An atom is said to be ground if none of the terms in the atom contain a variable

Definition 6 (Type Semantics)

Given an ASP Program Π , the semantics of Π is defined using:

- D_e = the set of terms and functions in the language of Π ;
- D_a = the set of atoms in the language of Π ;
- D_l = the set of literals;
- D_g = the set of gen-literals;
- D_h = the set of “or”-connected literals that belong to heads of ASP rules;
- D_d = the set of “,”-connected gen-literals that belong to bodies of ASP rules;
- $D_t = \{0, 1\}$, the set of satisfiability values for an ASP program; and
- $D_{a \rightarrow b}$ = the set of functions from D_a to D_b .

Expressions of type t denote satisfiability values of ASP programs. An ASP program can be true under certain Herbrand interpretations, and false under others. $(a \rightarrow b)$ denotes functions whose input is in D_a and output values are in D_b . For example, the type $(e \rightarrow t)$ corresponds to functions from terms to satisfiability values.

This section continues by introducing the definition for ASP typed term, followed by the definition of ASP λ -calculus formula:

Definition 7 (ASP Typed Term)

The elements which belong to the set Δ_A of ASP typed terms of type A are inductively defined as follows:

1. For each type A , every λ -term of type A belongs to Δ_A .
2. For any types A and B
 - if $\alpha \in \Delta_{A \rightarrow B}$ and $\beta \in \Delta_A$, then $\alpha @ \beta \in \Delta_B$
 - if u is a variable of type A and $\alpha \in \Delta_B$ has free occurrences of the variable u , then $\lambda u. \alpha \in \Delta_{A \rightarrow B}$ and the free occurrences of u are now bound to the abstractor λu .⁵
3. If f is a function symbol with arity n , and $t_1, t_2, \dots, t_n \in \Delta_e$, then $f(t_1, t_2, \dots, t_n) \in \Delta_e$.
4. If p is a predicate symbol with arity n , and $t_1, t_2, \dots, t_n \in \Delta_e$, then $p(t_1, t_2, \dots, t_n) \in \Delta_a$.
5. If $\alpha \in \Delta_a$ and α is not a λ -term⁶, then $\alpha \in \Delta_l$ and $(\neg \alpha) \in \Delta_l$.
6. If $\alpha \in \Delta_l$ and α is not a λ -term, then $\alpha \in \Delta_g$ and $(not \alpha) \in \Delta_g$.
7. If $\alpha \in \Delta_l$ and α is not a λ -term, then $\alpha \in \Delta_h$.
8. If $\alpha, \beta \in \Delta_h$, then $\alpha \text{ or } \beta \in \Delta_h$.
9. If $\alpha \in \Delta_g$ and α is not a λ -term, then $\alpha \in \Delta_d$.
10. If $\alpha, \beta \in \Delta_d$, then $\alpha, \beta \in \Delta_d$.
11. If $\alpha \in \Delta_h$ and $\beta \in \Delta_d$, then $(\alpha \leftarrow \cdot) \in \Delta_t$, $(\leftarrow \beta \cdot) \in \Delta_t$, and $(\alpha \leftarrow \beta \cdot) \in \Delta_t$.
12. If $\rho_1, \rho_2 \in \Delta_t$, then $(\rho_1 \rho_2) \in \Delta_t$.

⁵ Refer to the definition of occurrence presented at the end of this section.

⁶ This is to guarantee that each λ -term only corresponds to its unique type.

Definition 8 (Typed ASP λ -Calculus Formula)

A typed ASP λ -calculus formula is an ASP typed term where every variable is bound to an abstractor and every abstractor binds to a variable.

A typed ASP λ -calculus formula is in β -normal form if it does not contain any β -redex occurrences. An example of a β -redex is a typed term of the form $(\lambda v.v)@John$. The typed term *John* or $\lambda v.v$, do not have any β -redex occurrences (Hindley 1997).

The binding of every variable to an abstractor and every abstractor to a variable in definition 8 correspond to closed and λI -terms in the classic theory of lambda calculus, respectively. These conditions ensure that one obtains Answer Set Programming programs when the typed ASP λ -calculus formulas are in β -normal form. By the way in which the ASP typed terms have been defined and the two properties that we are enforcing, when there are no lambda abstractors left in an ASP typed term we obtain an expression that belongs to the Answer Set Programming language presented above. Some examples of typed ASP λ -calculus formulas are the following:

- $\lambda w.\lambda v.(w \leftarrow v@X.)$ with type $(h \rightarrow ((e \rightarrow d) \rightarrow t))$ where w has type h , v has type $(e \rightarrow d)$, and X has type e .
- $\lambda x.\lambda y.(\leftarrow h(x), \text{not } \neg y.)$ with type $(e \rightarrow (a \rightarrow t))$ where x has type e and y has type a .
- $\lambda v.(v \text{ or } \neg v \leftarrow \cdot)$ with type $(a \rightarrow t)$ where v has type a .
- $\lambda w.(\lambda u.(w@ \lambda v.(position(v, u))))$ with type $((e \rightarrow l) \rightarrow t) \rightarrow (e \rightarrow t)$ where w has type $((e \rightarrow l) \rightarrow t)$, u and v have type e .

Let the fourth formula of the examples be J . In J , w has type $((e \rightarrow l) \rightarrow t)$ because when an ASP typed formula is applied J , it will be placed in the variable w and will receive as argument the expression $\lambda v.(position(v, u))$. This expression has type $(e \rightarrow l)$ and therefore the input of the formula applied to J needs to have $(e \rightarrow l)$ as input and t as output to lead to an ASP formula. Thus, w has type $((e \rightarrow l) \rightarrow t)$. u has type e meaning that one expects a term to be placed inside the literal *position*.

The following are **not** typed ASP λ -calculus formulas:

1. $\lambda y.\lambda x.(y \text{ or } \text{not } x@X)$, where x has type $(e \rightarrow l)$ and y have type l .
2. $\lambda v.\lambda w.(\neg w \leftarrow \neg \text{not } v@X)$, where w has type a , v has type $(e \rightarrow l)$, and X has type e .

The first expression is not a λ -calculus formula since x is of type $(e \rightarrow l)$ which means $x@X$ has type l from r_2 above. Then $\text{not } x@X$ must have type g from r_5 . However, there is no rule that allows us to combine y , an element of type l , with $\text{not } x@X$, an element of type g , with the connective *or*.

The second expression violates the rules of a typed ASP λ -calculus formulas since there is no rule saying that the connective \neg can be applied to terms of type g , which in this example is the type of $\text{not } v@X$.

This section concludes with two more definitions.

Definition 9 (Occurrence)

The relation P occurs in Q is defined by induction on Q as follows:

- an ASP typed term P occurs in P .
- if P occurs in M or in N , then P occurs in $M@N$.
- if P occurs in M , then P occurs in $\lambda x \cdot M$.
- if P occurs in ϕ or P occurs in ψ , then P occurs in ϕ or ψ , $\phi \leftarrow \psi$ and ϕ, ψ .
- if P occurs in ϕ , then P occurs in $\neg\phi$ and *not* ϕ .
- if P occurs in any term t_i , then P occurs in $F(t_1, \dots, t_n)$. Where F is a function symbol.
- if P occurs in any term t_i , then P occurs in $R(t_1, \dots, t_n)$. Where R is a predicate symbol of an atom.

Definition 10 (sub-term)

A sub-term of a typed ASP λ -calculus formula F is any term P that occurs in F .

3.1 Type Order

We have introduced the types that will be assigned to typed ASP lambda calculus terms and formulas. Next, we present the notion of order, which is associated with types and establishes a hierarchical structure that separates typed λ -calculus formulas to several classes. Order will be an important concept when we state the completeness proof for the Inverse λ -Algorithms since we will show that they are complete for typed λ -calculus formulas up to order two. Each typed term has a type, and each type will be assigned an order.

Definition 11 (Type Order)

The order of a type is defined as:

1. Base types have order 0.
2. For function types, $order(a \rightarrow b) = \max(order(a) + 1, order(b))$.

The definition from (Stirling 2009) gives order one to base types. In our case, we consider order zero for base types since this is the common approach in linguistics. Next, some examples of typed ASP lambda calculus formulas with different orders are presented:

- Order zero: $bird(tweety) \cdot$ - type t .
- Order one: $\lambda v. \lambda u. (v \leftarrow u \cdot)$ - type $(h \rightarrow (d \rightarrow t))$.
- Order two: $\lambda v. \lambda u. (v@X \leftarrow u@X \cdot)$ - type $((e \rightarrow l) \rightarrow ((e \rightarrow g) \rightarrow t))$.
- Order three: $\lambda w. (w@(\lambda z. h(z)) \cdot)$ - type $((e \rightarrow l) \rightarrow t) \rightarrow t$.

With these simple examples, one can see the intuition behind the order of typed ASP lambda calculus formulas. Formulas of order zero correspond to expressions with base types. Formulas of order one correspond to expressions which start with a series of lambda abstractors followed by an ASP program with variables bound to the initial lambda abstractors.

Formulas of order two extend the expressions allowed in order one by including applications. Formulas of order zero can be applied to variables inside the formula. Formulas of order three extend those present in order two by allowing lambda abstractors inside the expression after the initial lambda abstractors. In this case, formulas of order one can be applied to variables, this is why now, we can find lambda abstractors at the beginning and in the middle of the formulas. These claims can be easily proved by contradiction using the given definitions.

4 The Inverse Lambda Operators

This section presents the formal definition of the two components of the Inverse λ -Algorithms, $Inverse_L$ and $Inverse_R$, from (Gonzalez 2010). The objective of $Inverse_L$ and $Inverse_R$ is that, given typed λ -calculus formulas H and G , the formula F is computed such that $F@G = H$ and $G@F = H$, respectively. We now define the different symbols used in the algorithm and their meaning:

- Let G , H and J represent typed λ -calculus formulas, J^1, J^2, \dots, J^n represent typed terms; v , w and v_1, \dots, v_n represent variables.
- Typed terms that are sub-terms of a typed term J^i are denoted as J_k^i .

We also consider the following two statements:

- A list of λ -abstractors of the form $\lambda v_1, \dots, v_i$ can be empty if the corresponding variables v_1, \dots, v_i are not present in the formula they belong to.
- If the formulas being processed within the algorithm do not satisfy any of the *if* conditions then the algorithm returns *null*.

Definition 12 (Operator :)

Consider two lists (of same length) of typed ASP λ -calculus formulas A_1, \dots, A_n and B_1, \dots, B_n , and a typed ASP λ -calculus formula H . The result of the operation $H(A_1, \dots, A_n : B_1, \dots, B_n)$ is defined as:

1. find the first occurrence of formulas A_1, \dots, A_n in H .
2. replace each A_i by the corresponding B_i .
3. find the next occurrence of formulas A_1, \dots, A_n in H and go to 2. Otherwise, stop.

We now give the two inverse algorithms.

Definition 13 ($Inverse_L(H, G)$)

The algorithm $Inverse_L(H, G)$, is defined as:

Given G and H :

1. If G is $\lambda v.v$
 - then $F = \lambda v.(v@H)$
2. If G is a sub-term of H
 - then $F = \lambda v.H(G : v)$

3. If G is not $\lambda v.v$, $(J^1(J_1^1, \dots, J_m^1), J^2(J_1^2, \dots, J_m^2), \dots, J^n(J_1^n, \dots, J_m^n))$ are sub-terms of H , and $\forall J^i \in H$, G is $\lambda v_1, \dots, v_s \cdot J^i(J_1^i, \dots, J_m^i : v_{k_1}, \dots, v_{k_m})$ ⁷ with $1 \leq s \leq m$ and $\forall p$, $1 \leq k_p \leq s$.
 - then $F = \lambda w.H((J^1 : (w @ J_{k_1}^1 @ \dots @ J_{k_m}^1), \dots, J^n : (w @ J_{k_1}^n @ \dots @ J_{k_m}^n)))$ where each J_{k_p} maps to a different v_{k_p} in G .
4. If H is $\lambda v_1, \dots, v_i \cdot J$ and $J^1(J_{i+1}^1, \dots, J_s^1)$ is a sub-term of J , G is $\lambda w.J(J^1(J_{i+1}^1, \dots, J_s^1) : w @ J_{k_1}^1 @ \dots @ J_{k_s}^1)$ with $\forall p$, $i+1 \leq k_p \leq s$.
 - then $F = \lambda w.\lambda v_1, \dots, v_i \cdot (w @ \lambda v_{i+1}, \dots, v_s \cdot (J^1(J_{i+1}^1, \dots, J_s^1 : v_{k_1}, \dots, v_{k_s})))$

Definition 14 ($Inverse_R(H, G)$)

The algorithm $Inverse_R(H, G)$, is defined as:

Given G and H :

1. If G is $\lambda v.v @ J$
 - then $F = Inverse_L(H, J)$
2. If J is a sub-term of H and G is $\lambda v.H(J : v)$
 - then $F = J$
3. If G is not $\lambda v.v @ J$, $(J^1(J_1^1, \dots, J_m^1), J^2(J_1^2, \dots, J_m^2), \dots, J^n(J_1^n, \dots, J_m^n))$ are sub-terms of H and G is $\lambda w.H((J^1(J_1^1, \dots, J_m^1) : w @ J_{k_1}^1 @ \dots @ J_{k_m}^1), \dots, (J^n(J_1^n, \dots, J_m^n) : w @ J_{k_1}^n @ \dots @ J_{k_m}^n))$ with $1 \leq s \leq m$ and $\forall p$, $1 \leq k_p \leq m$.
 - then $F = \lambda v_1, \dots, v_s.J^1(J_1^1, \dots, J_m^1 : v_{k_1}, \dots, v_{k_m})$.
4. If H is $\lambda v_1, \dots, v_i \cdot J$ and $J^1(J_{i+1}^1, \dots, J_s^1)$ is a sub-term of J , G is $\lambda w.\lambda v_1, \dots, v_i.(w @ \lambda v_{i+1}, \dots, v_s \cdot (J^1(J_{i+1}^1, \dots, J_s^1 : v_{k_1}, \dots, v_{k_s})))$ with $\forall p$, $i+1 \leq k_p \leq s$.
 - then $F = \lambda w \cdot J(J^1(J_{i+1}^1, \dots, J_s^1) : w @ J_{k_1}^1 @ \dots @ J_{k_s}^1)$

Please note that the final cases for both operators involve formulas of third order.

5 Inverse Lambda Algorithm Examples with Typed ASP Lambda Calculus

This section presents several examples demonstrating how the Inverse- λ Algorithms can be applied to find F in various settings given ASP- λ -calculus formulas G and H . A use case example follows in the next section.

5.1 Example 1

Let H and G be typed ASP λ -calculus formulas where $H = bird(tweety)$. and $G = \lambda x.x$. F needs to be calculated such that $H = F @ G$. Here case 1 of $Inverse_L$ will be applicable. Then $F = \lambda v.(v @ H)$ and in this case $F = \lambda v.(v @ bird(tweety))$. Then, $F @ G = \lambda v.(v @ bird(tweety)) @ \lambda x.x = (\lambda x.x @ bird(tweety)) = bird(tweety) = H$.

⁷ When the formula G is being generated, the indexes of the abstractors $\lambda v_1, \dots, v_s$ must be assigned to bind the variables from v_{k_1}, \dots, v_{k_m} in such a way that G is a valid formula.

5.2 Example 2

Let H and G be typed ASP λ -calculus formulas where $H = \lambda u.(\text{fly}(X) \leftarrow u, \text{not } \neg \text{fly}(X).)$ and $G = \text{fly}(X)$. F needs to be calculated such that $H = F @ G$. Here case 2 of $Inverse_L$ is applicable. Thus, we get $F = \lambda v.H(G : v) = \lambda v.H(\text{fly}(X) : v) = \lambda v.\lambda u.(v \leftarrow u, \text{not } \neg v.)$

5.3 Example 3

Let H and G be typed ASP λ -calculus formulas with $H = \lambda u.(\text{bird}(\text{tweety}), \text{animal}(\text{tweety}), \text{penguin}(\text{rocky}), \text{animal}(\text{rocky}), \text{eats}(\text{tweety}, u))$ and $G = \lambda v.\lambda w.(v, \text{animal}(w))$. F now needs to be calculated such that $H = F @ G$. Therefore, case 3 of $Inverse_L$ will be applicable.

G is not $\lambda v.v$ so the first condition is satisfied. From H , one has the following formulas that are subterms: $J^1 = \text{bird}(\text{tweety}), \text{animal}(\text{tweety})$ with sub-subterms $J_1^1 = \text{bird}(\text{tweety})$ and $J_2^1 = \text{tweety}$ (from $\text{animal}(\text{tweety})$); $J^2 = \text{penguin}(\text{rocky}), \text{animal}(\text{rocky})$ with sub-subterms $J_1^2 = \text{penguin}(\text{rocky})$ and $J_2^2 = \text{rocky}$ (from $\text{animal}(\text{rocky})$). Therefore the second condition of case 2 is satisfied.

The third condition is satisfied since, $\forall J^i \in H: G = \lambda v_1.\lambda v_2.J^i(J_1^i, J_2^i : v_1, v_2)$ for $i = 1, 2$. For example, for J^1 , $G = \lambda v_1.\lambda v_2.J^1(\text{bird}(\text{tweety}), \text{tweety} : v_1, v_2) = \lambda v.\lambda w.(v, \text{animal}(w))$.

Therefore one can now calculate that $F = \lambda w.H((J^1 : w @ J_1^1 @ J_2^1), (J^2 : w @ J_1^2 @ J_2^2)) = \lambda x.H((J^1 : x @ \text{bird}(\text{tweety}) @ \text{tweety}), (J^2 : x @ \text{penguin}(\text{rocky}) @ \text{rocky})) = \lambda x.\lambda u.(x @ \text{bird}(\text{tweety}) @ \text{tweety}, x @ \text{penguin}(\text{rocky}) @ \text{rocky}, \text{eats}(\text{tweety}, u))$.

5.4 Example 4

Let H and G be typed ASP λ -calculus formulas with $H = \text{love}(\text{Mia}, \text{Jon}) \leftarrow \text{love}(\text{Jon}, \text{Mia})$. and $G = \lambda w.w @ \text{Mia} @ \text{Jon} \leftarrow w @ \text{Jon} @ \text{Mia}$. F needs to be calculated such that $H = G @ F$. Case 3 of $Inverse_R$ will be applied.

G is clearly not $\lambda v.v @ J$. H has the following subterms: $J^1(J_1^1, \dots, J_m^1) = \text{love}(\text{Mia}, \text{Jon})$ with sub-subterms $J_1^1 = \text{Mia}$ and $J_2^1 = \text{Jon}$; $J^2(J_1^2, \dots, J_m^2) = \text{love}(\text{Jon}, \text{Mia})$ with $J_1^2 = \text{Jon}$ and $J_2^2 = \text{Mia}$. Then, $G = \lambda w.H((J^1(J_1^1, J_2^1) : w @ J_1^1 @ J_2^1), (J^2(J_1^2, J_2^2) : w @ J_1^2 @ J_2^2)) = \lambda w.(\text{love}(\text{Mia}, \text{Jon}) : w @ \text{Mia} @ \text{Jon}) \leftarrow (\text{love}(\text{Jon}, \text{Mia}) : w @ \text{Jon} @ \text{Mia}) = \lambda w.w @ \text{Mia} @ \text{Jon} \leftarrow w @ \text{Jon} @ \text{Mia}$. Therefore, G satisfies the second condition of case 3.

Thus, we calculate $F = \lambda v_1.\lambda v_2.J^1(J_1^1, J_2^1 : v_1, v_2) = \lambda v_1.\lambda v_2.(\text{love}(\text{Mia}, \text{Jon} : v_1, v_2)) = \lambda v_1.\lambda v_2.\text{love}(v_1, v_2)$.

5.5 Example 5

Let H and G be typed ASP λ -calculus formulas where $H = \lambda v.(\text{stay_at}(\text{room5}) \leftarrow \text{not goto_from}(v, \text{room5}).)$ and $G = \lambda w.\lambda v.(w @ \lambda u.\text{goto_from}(v, u))$. F needs to be calculated such that $H = G @ F$. Therefore, case 4 of $Inverse_R$ will be applied.

$H = \lambda v.J$ with $J = \text{stay_at}(\text{room5}) \leftarrow \text{not goto_from}(v, \text{room5}). f(\sigma_{i+1}, \dots, \sigma_s) = \text{goto_from}(v, \text{room5})$ with $s = 2$ and $\sigma_2 = \text{room5}$. Relabeling the variables of G

to better match the conditions of the case by substituting v_1 for v and v_2 for u , we see $G = \lambda w. \lambda v_1. (w @ \lambda v_2. (f(\sigma_2 : v_2)))$. Therefore, G satisfies the second condition of case 4.

Thus, we calculate $F = \lambda w. J(f(\sigma_2) : w @ \sigma_2) = \lambda w. (stay_at(room5) \leftarrow not\ w @ room5.)$

6 Use Case Examples

In this section we present a use case of our inverse lambda algorithms to show how meaning of words are computed when one knows meaning of the sentences and meaning of some of the words. We consider the following sentences from (Baral et al. 2008).

- Most birds fly.
- Penguins are birds.
- Penguins do not fly.

We will consider an initial lexicon that has the semantics for simple nouns and verbs. Combinatory Categorical Grammar (CCG) (Clark and Curran 2007) is used to construct the meaning of a sentence from the meaning of its constituent words and phrases. After parsing the first two sentences using CCG and adding the semantics from the initial lexicon, we obtain the output of a simplified CCG parsing with two categories “S” (sentence) and “NP” (noun phrase), as shown in Table 2.

	Most	birds	fly
	$(S/(S \backslash NP))/NP$	NP	$S \backslash NP$
	$S/(S \backslash NP)$		
	S		
Penguins	are	birds	
NP	$(S \backslash NP)/NP$	NP	
NP	$S \backslash NP$		
	S		
Most	birds	fly	
???	$\lambda x. bird(x)$	$\lambda x. fly(x)$	
???		$\lambda x. fly(x)$	
	$fly(X) \leftarrow bird(X), not \neg fly(X)$		
Penguins	are	birds	
$\lambda x. penguin(x)$???	$\lambda x. bird(x)$	
$\lambda x. penguin(x)$???		
	$penguin(X) \leftarrow bird(X)$		

Table 2. CCG and λ -calculus derivation for “Most birds fly” and “Penguins are birds”.

In Table 2, one can see that the semantic representations for the words “most” and “are” are missing. We already discussed how we can obtain the semantic representation of “most” before. Now we will illustrate how we can compute it using the presented Inverse λ -Algorithms. Starting with the first sentence, one can take the meaning of the sentence and the meaning of the word “fly” to calculate the representation of “Most birds”.

“Most birds” has category $S/(S \backslash NP)$ and the category of “fly” is being applied from its right. Therefore, if we take H as the meaning of the sentence and G as the meaning of “fly”, we can use $Inverse_L(H, G)$ to obtain the expression for “Most birds”. In this case, option three of the algorithm is satisfied and $F = \lambda x. (x @ X \leftarrow bird(X), not \neg x @ X)$.

Now, we have the expression for “Most birds” and “birds”. Since the word “birds” is being applied to the right of “Most birds”, we need to again call $Inverse_L(H, G)$ to obtain the representation for “Most”. Option three of the algorithm is again satisfied and one obtains F as $\lambda v.\lambda x.(x@X \leftarrow v@X, \text{ not } \neg x@X)$. This is the Typed ASP λ -calculus representation for the word “most”.

The process to obtain the word “are” from the second sentence is very similar. First one calls $Inverse_L(H, G)$ with H being the meaning of the sentence and G being “Penguins” to obtain the meaning of “are birds”. Option three of the algorithm is satisfied and $F = \lambda x.(x@X \leftarrow \text{bird}(X))$. Next, one calls $Inverse_L(H, G)$ with “are birds” and “birds” to obtain the desired meaning of “are”. Option three of the algorithm is satisfied again and $F = \lambda v.\lambda x.(x@X \leftarrow v@X)$. This expression corresponds to the Typed ASP λ -calculus formula for the word “are”. Next, the last sentence is presented in Table 3.

Penguins	do not	fly
$\frac{NP}{(S/(S \setminus NP))}$	$\frac{(S/(S \setminus NP)) \setminus NP}{S \setminus NP}$	$S \setminus NP$
S		
Penguins	do not	fly
$\lambda x.\text{penguin}(x)$	$???$	$\lambda x.\text{fly}(x)$
$???$		
$\neg \text{fly}(X) \leftarrow \text{penguin}(X)$		

Table 3. CCG and λ -calculus derivations for “Penguins do swim” and “Penguins do not fly”.

In this case, the semantics of the phrase “do not” is missing. This phrase was not part of the initial lexicon. For this sentence, we call $Inverse_L$ first, obtaining $\lambda x.(\neg x@X \leftarrow \text{penguin}(X))$ as the meaning of “Penguins do not”, and $Inverse_L$ afterwards to obtain the representation for “do not”, which is $\lambda u.\lambda x.(\neg x@X \leftarrow u@X)$.

7 Correctness and Complexity of the Inverse Algorithms

Theorem 1 (Soundness of $Inverse_L$)

Given two typed λ -calculus formulas H and G in β -normal form, if $Inverse_L(H, G)$ returns a non-null value F , then $H = F @ G$.

Theorem 2 (Soundness $Inverse_R$)

Given two typed λ -calculus formulas H and G in β -normal form, if $Inverse_R(H, G)$ returns a non-null value F , then $H = G @ F$.

Theorem 3 (Completeness of $Inverse_L$)

For any two typed λ -calculus formulas H and G in β -normal form, where H is of order two or less, and G is of order one or less, if there exists a set of typed λ -calculus formulas Θ_F of order two or less in β -normal form, such that $\forall F_i \in \Theta_F, H = F_i @ G$, then $Inverse_L(H, G)$ will give an F where $F \in \Theta_F$.

Theorem 4 (Completeness of $Inverse_R$)

For any two typed λ -calculus formulas H and G of order two or less in β -normal form, if there exists a set of typed λ -calculus formulas Θ_F of order one or less in β -normal form, such that $\forall F_i \in \Theta_F, H = G@F_i$, then $Inverse_R(H, G)$ will give an F , where $F \in \Theta_F$.

Theorem 5 ($Inverse_L$ complexity)

The $Inverse_L$ Algorithm runs in exponential time in the number of variables in G and polynomial time in the size of the formulas H and G .

Theorem 6 ($Inverse_R$ complexity)

The $Inverse_R$ Algorithm runs in exponential time in the number of variables in G and polynomial time in the size of the formulas H and G .

Due to lack of space, we will only comment on how the soundness and completeness proofs are structured. The complete proofs are given in the online appendix of the paper. The soundness proof shows how in each of the four cases of $Inverse_L$, the typed ASP λ -calculus formula H is obtained by applying F to G . The application $F@G$ is computed using the expressions from the algorithm for F and G , generating the expression for H given in the algorithm. The proof of Theorem 1 is given in the online appendix of the paper, pp. 2–3. The same reasoning is followed for $Inverse_R$. The complete proof of Theorem 2 is given in the online appendix of the paper, pp. 3–4.

The completeness proof is divided to six cases, which correspond to the six possible valid combinations of orders that H , F and G may have, such that the order of the terms will be less than 2. These are shown in Table 4. For each case, it is proven by contradiction that $Inverse_L$ and $Inverse_R$ return a formula F if one such F exists. It is done by assuming that they return a *null* value and reaching a contradiction at the end of the proof. In the process, each of the four conditions of the algorithms are analyzed, where it is shown that at least one of the conditions of the algorithm has be satisfied for each of the six cases. The complete proof of Theorem 3 is given in the online appendix of the paper, pp. 4–7 and Theorem 4 is given in pp. 7–11.

Finally, the proof for the complexity results, Theorem 5 and Theorem 6, are given in the online appendix of the paper, pp. 11–12.

H	F	G	ASP type examples for formula F
0	1	0	$e \rightarrow t$
1	1	0	$a \rightarrow (e \rightarrow t)$
2	2	0	$d \rightarrow ((g \rightarrow t) \rightarrow t)$
0	2	1	$(h \rightarrow t) \rightarrow t$
1	2	1	$(l \rightarrow t) \rightarrow (e \rightarrow t)$
2	2	1	$(g \rightarrow t) \rightarrow ((e \rightarrow t) \rightarrow t)$

Table 4. Possible order combinations for F, G and H formulas, with $H = F@G$.

8 Conclusion

In this paper we developed the language of typed answer set programming lambda calculus and defined associated notions such as ASP typed term, ASP λ -calculus formulas, and type orders. We used these notions to formulate soundness and completeness of Inverse λ -Algorithms with respect to typed answer set programming lambda calculus. These algorithms are important in that they allow automatic construction of ASP-lambda representations of new words using information already available about known sentences and words. They have been used in a system that is able to learn to translate combinatorial logic puzzle descriptions to ASP rules (Baral and Dzifcak 2012) that obtain solutions to the puzzles; however that (short) paper does not go into the details of the algorithm, as we do here.

Acknowledgement

We acknowledge support by NSF (grant number 0950440), IARPA and ONR-MURI for this work.

References

- BARAL, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press.
- BARAL, C. AND DZIFCAK, J. 2012. Solving puzzles described in english by automated translation to answer set programming and learning how to do that translation. In *KR (to appear)*.
- BARAL, C., DZIFCAK, J., AND SON, T. C. 2008. Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*. 818–823.
- BARAL, C., GONZALEZ, M. A., AND GOTTESMAN, A. 2012. The inverse lambda calculus algorithm for typed first order logic lambda calculus and its application to translating english to fol. In *Logic-based Artificial Intelligence*, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds. Lecture Notes in Computer Science. Springer. To be published.
- BARBARA H. PARTEE, A. T. M. AND WALL, R. E. 1990. *Mathematical Methods in Linguistics*. Kluwer Academic Publishers.
- BARENDREGT, H. 1992. *Lambda Calculi with Types, Handbook of Logic in Computer Science*. Vol. II. Oxford University Press.
- BLACKBURN, P. AND BOS, J. 2005. *Representation and Inference for Natural Language: A First Course in Computational Semantics*. Center for the Study of Language.
- CHURCH, A. 1940. A formulation of the simple theory of types. *The Journal of Symbolic Logic* 5, 2, 56–68.
- CLARK, S. AND CURRAN, J. R. 2007. Wide-coverage efficient statistical parsing with ccg and log-linear models. *Computational Linguistics* 33.
- COSTANTINI, S. AND PAOLUCCI, A. 2010. Towards translating natural language sentences into asp. In *Proc. of the Intl. Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*.
- DOWEK, G. 1994. Third order matching is decidable. *Annals of Pure and Applied Logic* 69, 2-3, 135–155.
- DZIFCAK, J., SCHEUTZ, M., BARAL, C., AND SCHERMERHORN, P. 2009. What to do and how to do it: translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *Robotics and Automation, 2009. ICRA '09*. 4163–4168.
- GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*. 1070–1080.
- GONZALEZ, M. A. 2010. An Inverse Lambda Calculus Algorithm For Natural Language Processing, M.S thesis. Arizona State University.
- HINDLEY, J. 1986. *Introduction to Combinators and Lambda-Calculus*. Cambridge University press.
- HINDLEY, J. 1997. *Basic Simple Type Theory*. Cambridge University press.
- HUET, G. 1973. The undecidability of unication in third order logic. *Information and Control* 22, 3, 257–267.
- HUET, G. 1975. A unication algorithm for typed calculus. *Theoretical Computer Science* 1, 27–57.
- KWIATKOWSKI, T., ZETTLEMOYER, L., GOLDWATER, S., AND STEEDMAN, M. 2010. Inducing Probabilistic CCG Grammars from Logical Form with Higher-order Unification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1223–1233.

- LOADER, R. 2003. Higher-order beta-matching is undecidable. *Logic Journal of IGPL* 11, 1, 51–68.
- MONTAGUE, R. 1974. *Formal Philosophy. Selected Papers of Richard Montague*. New Haven: Yale University Press.
- STEEDMAN, M. 2000. *The syntactic process*. MIT Press.
- STIRLING, C. 2009. Decidability of higher-order matching. *To appear Logical Methods in Computer Science* 5, 3.
- ZETTLEMOYER, L. AND COLLINS, M. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *21th Annual Conference on Uncertainty in Artificial Intelligence*. 658–666.

This figure "notTree.JPG" is available in "JPG" format from:

<http://arxiv.org/ps/1210.5670v1>

This figure "positionTree.JPG" is available in "JPG" format from:

<http://arxiv.org/ps/1210.5670v1>